SOLUTIONS FOR OPTIMIZING THE DATA PARALLEL PREFIX SUM ALGORITHM USING THE COMPUTE UNIFIED DEVICE ARCHITECTURE

Ion Lungu[1]
Dana-Mihaela Petroşanu[2]
Alexandru Pîrjan[3]

Abstract

In this paper, we analyze solutions for optimizing the data parallel prefix sum function using the Compute Unified Device Architecture (CUDA) that provides a viable solution for accelerating a broad class of applications. The parallel prefix sum function is an essential building block for many data mining algorithms, and therefore its optimization facilitates the whole data mining process. Finally, we benchmark and evaluate the performance of the optimized parallel prefix sum building block in CUDA.

Keywords: CUDA, threads, GPGPU, parallel prefix sum, parallel processing, task synchronization, warp.

1. Introduction

The latest generations of GPU (Graphics Processing Units) architectures are much easier to program than traditional GPUs and offer a significant increase in both memory bandwidth and computational power; so many software developers have focused their attention lately on General Purpose Computation Graphics Processing Units (GPGPU). A GPGPU is particularly useful in many scientific fields (medical data analysis, neuroscience, telecommunication control, data extraction, financial data prediction) due to the huge computational power that goes far beyond than that of a CPU (Central Processing Unit). Graphics processing units combine hundreds of simplified parallel processing cores so the necessary time for data extraction is considerably reduced and this is very useful when performing operations on massive data workloads. This high computational power that overcomes the most powerful CPUs, offers a huge benefit for numerous scientific fields like image processing, geometric processing and database operations. Another essential aspect when comparing a GPGPU to a classical central processing unit is the performance per watt consumed. General-purpose computation graphics processing units prove to be powerful instruments, with high performance, low cost and an increasing number of features offered, capable of solving an increasingly wide range of applications [1].

For a long time, GPU processors have been used to accelerate graphics rendering on computers. Following the increasing need for improved three-dimensional rendering at a high resolution and a large number of frames per second, the GPU has evolved through a specialized architecture, from one-purpose components to multiple purposes complex

[1] PhD Professor, Economic Informatics Department, Academy of Economic Studies, e-mail: ion.lungu@ie.ase.ro
2 PhD, Department of Mathematics-Informatics I, University Politehnica of Bucharest,
e-mail: danap@mathem.pub.ro
[3] Assistant Lecturer, PhD Candidate, School of Computer Science for Business Management, Romanian-American University, e-mail: alex@pirjan.com

architectures, able to do much more than just provide video rendering. This development allowed the acceleration of a broad class of applications.

CUDA is a software and hardware architecture that allows the NVIDIA graphics processor to execute programs written in C, C++, FORTRAN, OpenCL, Direct Compute and other languages. A CUDA program calls parallel program kernels. A kernel executes in parallel a set of parallel threads. The programmer or compiler organizes these threads into thread blocks and grids of thread blocks. The GPU processor instantiates a kernel program on a grid containing parallel thread blocks. Each thread from the block executes an instance of the kernel and has an unique ID associated to registers, to thread's private memory from the thread block [2].

The CUDA programming environment provides three levels of abstraction [2]: a hierarchy of thread groups, shared memories and barrier synchronization. These abstractions, available for the developer as a minimal set of extensions in the C language, provide fine-grained parallelism for data and threads, associated with large grained parallelism for data and tasks. Using these abstractions, the developer partitions the problem into sub-problems of small sizes that can be solved in parallel. Such an approach allows threads to cooperate when solving each sub-problem and also provides scalability since each sub-problem can be solved by any of the available processing cores. Consequently, a CUDA C compiled program can be executed on any number of processing cores.

The Compute Unified Device Architecture hierarchy of threads is mapped to the hierarchy of the graphics processing units hardware processor; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; the CUDA cores contained in the SM run the threads within blocks. The streaming multiprocessor can perform up to 32 groups of threads called warps. Regarding memory hierarchy, each multiprocessor contains a set of 32-bit registry with a zone of shared memory, which is easily accessible for each core of the multiprocessor but hidden from other multi-processors. Depending on the generation of a GPU, the number of registry and the size of shared memory vary. Besides shared memory, a multiprocessor contains two read-only memory cache, one for texture and another one for constants.

With the introduction of CUDA-C language, application developers are able to harness, using a standard programming language, the huge parallel processing power offered by the latest generation of graphics processing units. CUDA enables developers to specify how tasks are decomposed in order to be processed by many parallel threads and how are these tasks executed by the GPU [3]. The high level of control offered by the CUDA-C language facilitates the development of high-performance basic functions useful for optimizing a wide range of computational tasks that require high processing power. Thus, CUDA-C is a viable solution for developing essential building blocks for many data mining algorithms and among them, the parallel prefix sum function.

The parallel prefix sum function offers to the developers the possibility of implementing various efficient algorithms for extracting and processing data so the programmer does not have to know in detail how resources are allocated in the architecture or to spend extra time in optimizing the parallel prefix sum function that is used in the algorithms' construction. The

parallel prefix sum function automatically allocates resources, by optimally taking into account the available hardware specifications. Therefore, the developer does not have to specify (but he has the possibility to do so) further restrictions or details regarding the division of tasks among multiple threads or their allocations to the processing cores, since the data parallel prefix sum function automatically specifies all these aspects.

By analyzing various implementations of data mining algorithms, we have noticed the possibility of improving their performance by optimizing the parallel prefix sum function that is a common functional building block in all of them. The function is developed to provide improved performance over other implementations. Another basic criterion for building this function is the modularity, so that the function can be easily included in other applications. The implementation of the parallel prefix sum function offers a series of advantages for the developers because, on one hand it provides a useful optimized tool in the data mining process and on the other hand, it offers significant economic advantages, much better than the CPU or even the GPU implementations.

Graphic Processing Units (GPU) are based on multiple processing cores and comprise thousands of parallel threads, allowing the parallel processing of data streams. The same program processes all data and the GPU uses the Single Instruction, Multiple Threads (SIMT) processing model. Thus, each processing core sequentially executes a thread and all the cores within the same group execute the same instruction in the same time. Instructions are processed in groups of 32 threads, called warps. The parallelism involved through the SIMT model is different from the task parallelism and the instruction-level parallelism. The usage of this type of parallelism in data processing algorithms offers optimal performance when algorithms are being designed to use the parallel processing capability of the architecture and relies on a good knowledge of the algorithm's complexity and on the implemented level of parallelism.

In this context, the data mining and its parallel processing improves by using essential building blocks for data mining algorithms (and among them, the parallel prefix sum) designed so as to provide optimum performance and efficiency, based on graphic processing units with multiple cores. One of the function's main advantages is the ability to reuse the source code in developing a wide range of algorithms for extracting and processing data. The ability to reuse the source code has a wide range of applications, allowing the possibility to optimize the data mining process as a whole, not just a specific application.

2. The data parallel prefix sum function

The prefix sum is an algorithmic function used in sorting routines, sequence compaction, radix sort, quicksort, sparse-matrix vector multiplication, minimum spanning tree construction, in graph theory and in many parallel algorithms [4].
Given an associative binary operator $\otimes$ with identity e and an input n-dimensional sequence $v = [a_0, a_1, \dots, a_{n-1}]$, a prefix sum operation produces an output n-dimensional sequence w_e, where

$$w_e = [e, a_0, (a_0 \otimes a_1), (a_0 \otimes a_1 \otimes a_2), \dots, (a_0 \otimes a_1 \otimes \dots \otimes a_{n-2})] \quad (1)$$

In the serial implementation of the prefix sum function, a single thread that uses a central processing unit (CPU) performs the computations. All the elements of the input vector are processed in an ascending order and the current element of the output vector is computed as the sum of the previous element of the output vector and the previous element of the input vector.

In the following, we present an efficient method for designing and implementing the parallel prefix sum function in the CUDA architecture. We have designed the function using a "divide and conquer" approach, based on a set of prefix sum functions built within the warps (groups of 32 threads). This approach results in an optimized, efficient function and this avoids multiple data requests from the same memory banks and, thus, the possibility of memory banks conflicts. The CUDA programming model, based on a hierarchy of threads, of thread blocks and grids of thread blocks [5], determines a hierarchical structure of the parallel prefix sum function. Within the algorithmic parallel prefix sum function, sequential computations are processed within each thread; results are parallel processed within each thread block and also between different blocks. In order to design an optimized prefix sum function, we have taken into account the parallel processing that takes place within each thread block and between different blocks, using shared memory management optimization techniques of the graphics processor as to avoid memory bank conflicts. In order to reach the maximum efficiency we have developed the parallel prefix sum function so that it fits the way how tasks are scheduled within the CUDA architecture. Within a warp, the parallel prefix sum function processes the instructions and then, the results are being processed using instructions at the thread blocks' level. Finally, the results obtained at the grids of thread blocks level are combined and thus it is obtained a global sum.

First we define a parallel prefix sum function within a warp consisting of 32 threads. (**Figure 1**).

```
template <class Operatorul,TipScanare Tip,class T>
__device__ T SumareWarp(volatile T *ptr,const unsigned int index= threadIdx.x)
{
    const unsigned int IndexFir = index & 31;
    if (IndexFir >= 1)
        ptr[index] = Operatorul::proceseaza(ptr[index - 1],ptr[index]);
    if (IndexFir >= 2)
        ptr[index] = Operatorul::proceseaza(ptr[index - 2],ptr[index]);
    if (IndexFir >= 4)
        ptr[index] = Operatorul::proceseaza(ptr[index - 4],ptr[index]);
    if (IndexFir >= 8)
        ptr[index] = Operatorul::proceseaza(ptr[index - 8],ptr[index]);
    if (IndexFir >= 16)
        ptr[index] = Operatorul::proceseaza(ptr[index - 16],ptr[index]);
    if(Tip == inc)
        return ptr[index ];
    else
        return (IndexFir >0)? ptr [index -1] : Operatorul :: ElementNeutru();
}
```

Figure 1. The parallel prefix sum function within a warp

In order to obtain an optimized implementation of the function, one must take into account [4] that threads within a warp are executed synchronously, eliminating the necessity of synchronization. Since a warp has 32 threads, it is not necessary to implement a "for" loop when browsing it.
The parallel prefix sum algorithmic function within the thread block uses the prefix sum function within a warp to process in parallel all warps sums. Each of the warps obtains a partial result and the last partial result of each warp is stored. A certain warp applies the warp level function "SumareWarp" on the previously stored results. Each of the warp's threads stores the partial results and then the final results are obtained as in **Figure 2**. The above described block level parallel prefix sum function has taken into account that the binary operator is associative.

```
template < class Operatorul, TipScanare Tip, class T>
__device__ T SumareBloc ( volatile T *ptr, const unsigned int index= threadIdx.x)
{
    const unsigned int IndexFir = index & 31;
    const unsigned int IdWarp = index >> 5;
    // Etapa 1. Se procesează în paralel sumele la nivelul warp-urilor folosind SumareWarp
    T rez = SumareWarp <Operatorul,Tip>(ptr,index);
    __syncthreads ();
    // Etapa 2. Se stochează ultimul rezultat parţial obţinut de fiecare dintre warp-uri.
    if(IndexFir ==31) ptr[IdWarp ] = ptr[index];
    __syncthreads ();
    // Etapa 3. Primul warp insumeaza rezultatele obtinute in etapa a 2-a.
    if(IdWarp ==0 ) SumareWarp <Operatorul,inclusiv>(ptr,index);
    __syncthreads ();
    // Etapa 4: Se acumuleaza rezultatele din etapele 1 si 3
    if (IdWarp > 0) rez = Operatorul::proceseaza(ptr[IdWarp -1],rez);
    __syncthreads ();
    // Step 5: Se scriu si se returneaza rezultatele finale
    ptr[index] = rez;
    __syncthreads ();
    return rez ;
}
```

Figure 2. The parallel prefix sum function within a thread block

We present next a parallel prefix sum function at the global level, for sequences of arbitrary length. This function uses the above mentioned block level function "SumareBloc", that performs a fixed size sum, corresponding to the thread block's size. The partial result for each of the thread blocks is stored in the array "rezultate_bloc[]". The parallel prefix sum function is applied to this storage array and each thread of the block stores the previous result, thus obtaining the final result.

3. The experimental results and the performance analysis for the parallel prefix sum function

In this section we analyze the performance of the above described parallel prefix sum function, using the following configuration: Intel i7-2600K operating at 3.4 GHz with 8 GB (2x4GB) of 1333 MHz, DDR3 dual channel. Programming and access to the GPUs used the CUDA toolkit 4.0, with the NVIDIA driver version 270.81. In addition, all processes related to graphical user interface have been disabled to reduce the external traffic to the GPU. The NVIDIA graphic cards used were the GeForce GTX 280 and the GTX480 (from the Fermi architecture). In [1] there are described the main features of these graphic cards.

Measurements do not include the necessary time for data transfers between the central processing unit and the graphic processing unit, as the parallel prefix sum is designed to be used as a building block for a large number of applications running on GPUs, so the transfer times will vary depending on the complexity of the specific application.
In order to compute the average execution time that the GPU spends for executing the parallel prefix sum function, we used the CUDA event application programming interface (API). We have preferred this option instead of those based on the CPU's or the operating system's timers, because those methods would have included latency and variations from different sources. In addition, we can asynchronously perform computations on the host while the GPU kernel runs and the only way to measure the necessary time for the host computations is to use the CPU or the operating system timing mechanism.

A GPU time stamp recorded at a user specified moment in time represents an event in CUDA. Because the time stamp is recorded directly by the GPU, we do not encounter the problems that could have appeared if we had tried to time the GPU execution using CPU timers. In order to time correctly the execution of the function, we create both a start and a stop event. Some of the kernel calls we make in CUDA C are asynchronous, the GPU begins to execute the code but the CPU continues the execution of the next code line before the GPU has finished. In order to safely read the value of the stop event we instruct the CPU to synchronize on the event using the API function "cudaEventSynchronize()", as depicted in

Figure 3.

```
float TimpulTotal = 0;
float timpul = 0;
cudaEvent_t inceput, sfarsit;
cudaEventCreate(&inceput);
cudaEventCreate(&sfarsit);
cudaEventRecord(inceput, 0);
//……….
//Functia algoritmica de baza al carei timp de executie il masuram
//……….
cudaEventRecord(sfarsit, 0);
cudaEventSynchronize(sfarsit);
//calculam timpul ce a trecut de la evenimentul de inceput pana la cel de sfarsit
CUDA_SAFE_CALL( cudaEventElapsedTime(&timpul, inceput, sfarsit));
TimpulTotal += timpul;
```

Figure 3. Measuring the execution time using CUDA events

In this way we instruct the runtime to block further instructions until the GPU has reached the stop event so when calling the "cudaEventSynchronize()" we are sure that all the GPU work prior to the stop event has been completed and it is safe to read the recorded time stamp. In this way, we get a reliable measurement of the execution time for computing the above described parallel prefix sum function [2].

The first set of tests evaluates the execution times obtained applying the parallel prefix sum function on vectors of various sizes and elements of float type. The vectors were randomly generated, as to cover a wide range of values. In **Table 1** we present the results of the experimental runs of the parallel prefix sum function over the CPU and the two GPUs

mentioned above. The results represent the average of 10,000 iterations, and the the unit of measure is milliseconds (ms).

Test No.	**Number of elements**	**Execution time on CPU (ms)**	**Execution time on GPU (ms)**	
			GeForce GTX280	**GeForce GTX480**
1	35	0.004226	0.081268	0.039465
2	128	0.000604	0.076837	0.03618
3	256	0.001509	0.081471	0.039502
4	260	0.001207	0.0786	0.036823
5	512	0.002113	0.081163	0.036878
6	1000	0.003924	0.07973	0.040742
7	1024	0.003622	0.078468	0.035371
8	1030	0.004226	0.239555	0.095693
9	32768	0.108366	0.249818	0.107071
10	45555	0.149117	0.251118	0.105261
11	65536	0.21462	0.24536	0.100109
12	131072	0.428636	0.274529	0.118679
13	262144	0.85516	0.285939	0.140495
14	500111	1.623384	0.339683	0.168711
15	524288	1.708206	0.349859	0.168474
16	1048555	3.439655	0.444213	0.227186
17	1048576	3.44207	0.42808	0.224189
18	1048581	3.45022	0.597514	0.285429
19	2097152	6.890177	0.791959	0.398717
20	2097999	6.860897	0.827059	0.39679
21	4194334	13.701268	1.127283	0.626898
22	8388600	27.653078	1.814563	1.076924

Table 1. Experimental results for the parallel prefix sum function

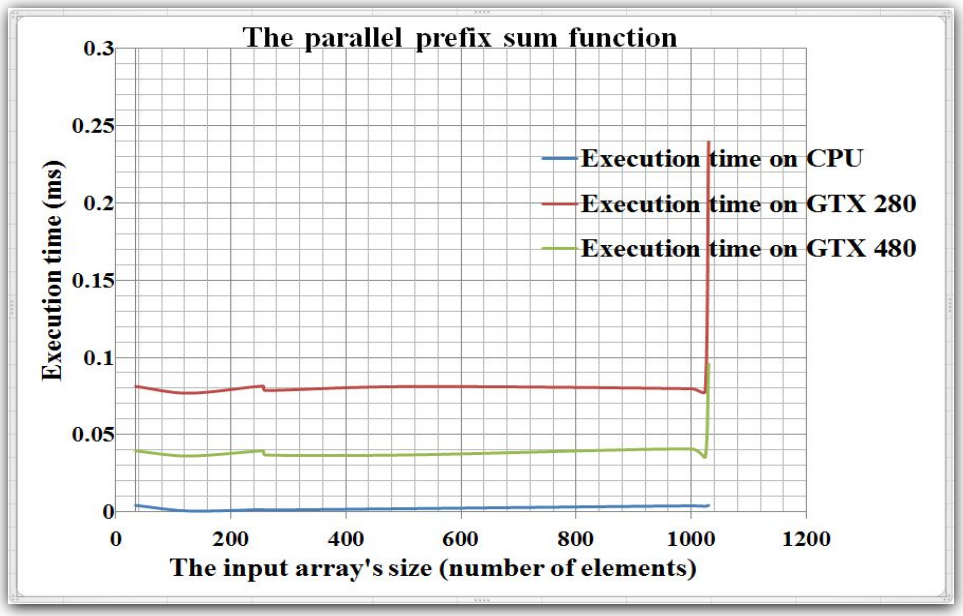

Figure 4. The parallel prefix sum function: 35-1030 elements of the input array

In **Figure 4** we present the obtained experimental results by running the parallel prefix sum function when the input array has a relatively low dimension (35-1030 elements). In this case we have noticed that the central processing unit has the best execution time, because it has not been generated an enough volume of computations in order to use the huge parallel processing capacity of the GPU.

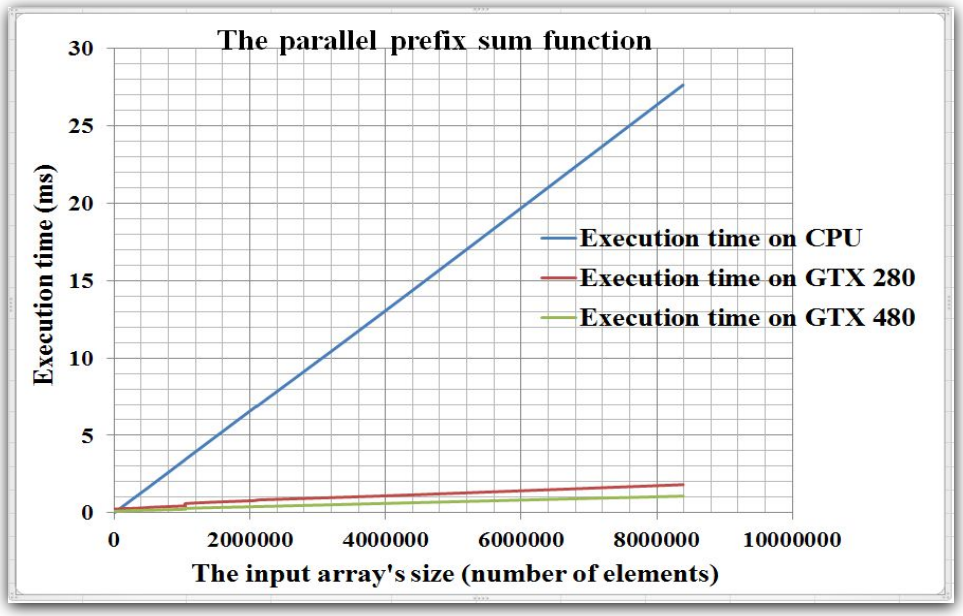

Figure 5. The parallel prefix sum function: 1030-8388600 elements of the input array

In **Figure 5** there are presented the obtained experimental results when running the parallel prefix sum function on a large dimension input array (1030-8388600 elements). In this case we have noticed that the best execution time is obtained by the GTX480 graphic card, because this time it has been created a sufficient computational load to fully employ the huge parallel processing capacity of the GPU. The CUDA implementation offers a high degree of performance whether the vector's dimension is a power of two or it is not.

The next set of tests evaluates the influence of data types on the performance of the above described parallel prefix sum function. The function has been designed to allow the selection of the input array components, which can be from one of the types: integer, unsigned integer, float, double, long long or unsigned long long. In **Figure 6** there are presented the obtained experimental results when running the parallel prefix sum function on an input array of variable dimension (35-8388600 elements). The results represent the average of 10,000 iterations. One can observe that the performance is comparable when the input data is integer, unsigned integer or float type, the execution time ranging between 0.033475 ms and 1.076924 ms. In the case when the input data is double, long long or unsigned long long type, the performance is comparable but the execution times are generally higher than in the three previous cases, ranging from 0.036127 ms and 2.142949 ms. This is justified taking into account the amount of memory necessary to store the analyzed data types.

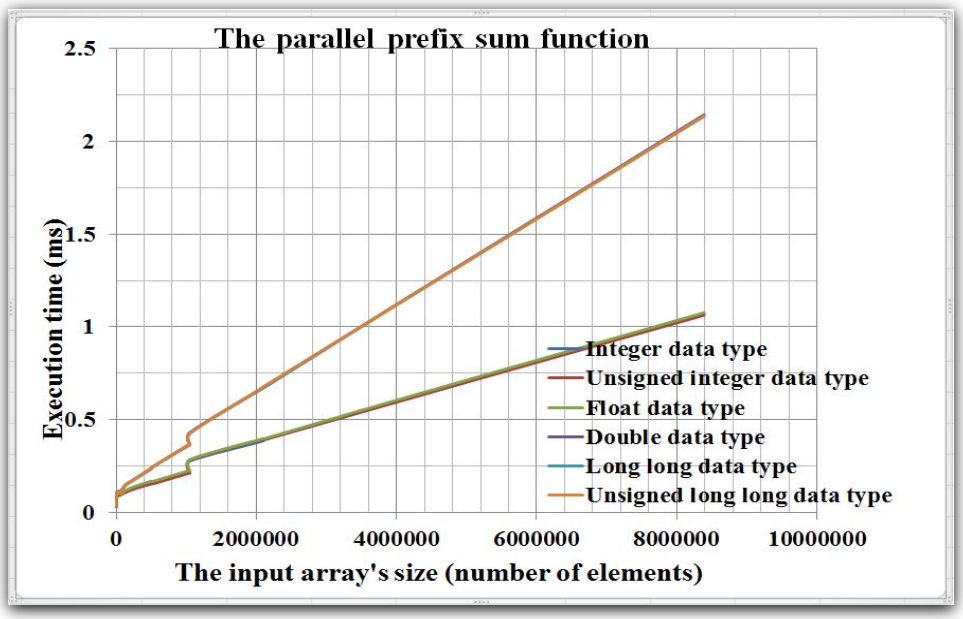

Figure 6. The influence of data types on the performance of the parallel prefix sum function

We analyze next the influence of the associative binary operator used when defining the parallel prefix sum function through equation (1), on the performance of the function. This function has been designed to allow the selection of the binary operator that can be one of the following: summation, maximum, minimum or multiplication.

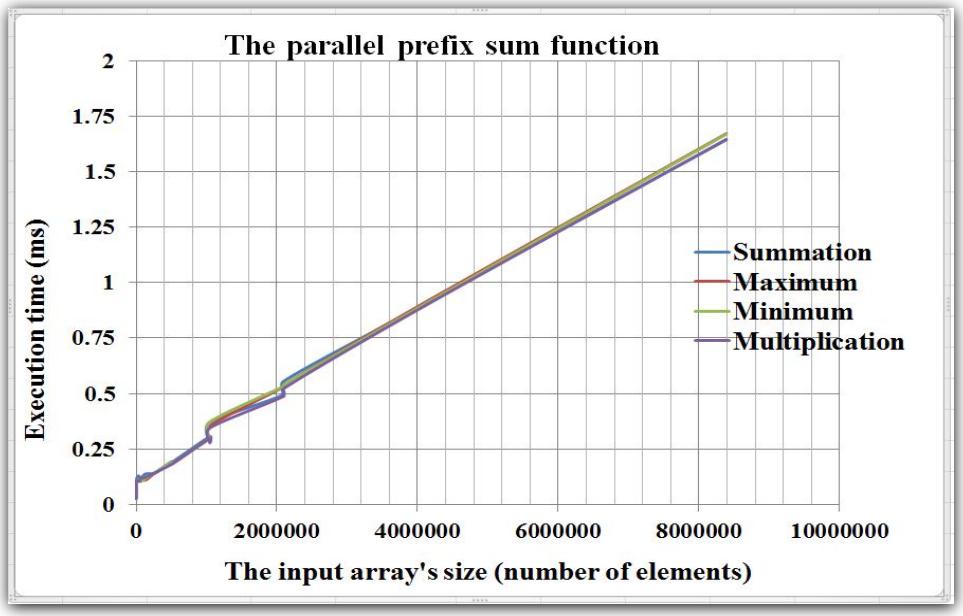

Figure 7. The influence of the associative binary operator on the performance of the parallel prefix sum function

In **Figure 7** there are presented the experimental results when running the parallel prefix sum function on an input array of variable dimension (35-8388600 elements). We have chosen float type elements for the input array. The results represent the average of 10,000 iterations. One can observe that the performance is comparable in all four cases of binary operators, the execution times ranging between 0.034305 ms and 1.090579 ms. Our experimental results confirm the function's efficiency, which offers optimum results in different situations and thus it can be implemented in a wide range of algorithms, without being influenced by the chosen binary operator.

-

4. Solutions for optimizing the performance of the parallel prefix sum function in CUDA

In the following, we outline a series of factors that influence the performance of the parallel prefix sum function in CUDA. The most important factor in improving the efficiency of the above-described function is related to the optimization of the tasks' assignment to each thread. Processing a single element per thread does not generate enough computational load to reduce the memory latency. Each thread has its own private memory and also has private registers, a program and a thread state counter. Each thread independently processes its source code. The GPU executes and manages at the hardware level hundreds of concurrent threads, avoiding the overloading and reducing the memory latency.

The Fermi architecture offers 512 processing CUDA cores; a GTX480 has 480 processing CUDA cores available for use. In order for the cores to be completely occupied, hundreds of threads are needed. When a defining the threads in order to improve the software performance in CUDA, the high latency of global memory is an important technical issue that must be

taken into account by the developer. CUDA generates and uses thousands of active threads, in contrast with the CPU designs that use large caches to hide memory latencies [1].

Thus, in the parallel prefix sum function in CUDA we have assigned eight input elements per thread. This happens when data is loaded from the global memory into the shared memory. Each thread reads two groups of four elements and then sequentially sums these groups. The obtained result in each of the four elements group is used as input data in a block-level function similar to the one described in **Figure 2**, the difference to the above described function being that each thread processes two input elements instead of one. When the block level function execution had finished, the result would have been stored back into the groups of four elements, that had been sequentially scanned.

Another method of optimization is to reduce the number of used registers. The GPU's architecture uses multithreading to reduce the memory latency, but the number of executing threads that can be simultaneously used is often limited by their registry requirements. Therefore, an efficient technique for maximizing the available number of threads is to minimize the number of used registers.

The parallel task synchronization represents real time coordination and is often associated with communication between threads. Synchronization is usually implemented by setting a synchronization point in the application from which a task cannot continue until another task reaches a certain point. The implementation of the parallel prefix sum function within a warp described in **Figure 1** eliminates the necessity of synchronization points, because communication between threads occur at the warp level. Unlike our implementation, a number of previous implementations [4] require the existence of synchronization points. The synchronization is not required for sharing data within the same warp, unlike the case when the shared data belongs to different warps. In order to process a parallel prefix sum, warps write their last element in a shared memory array, and then a single warp sums the previously stored results. Finally, each thread adds the warp's sum to its partial result.

The reduction of the used parallel steps is another factor that was taken into account for obtaining a high performance parallel prefix sum function compared with other similar functions. Although this had the effect of increasing the volume of necessary computations, the SIMT processing model of warps' execution facilitated their processing. In [4] there are described two versions of the parallel prefix sum function, one with a low efficiency regarding the volume of necessary computations and a second improved one, with a reduced number of computations.

The parallel prefix sum function has been optimized based on the usage of multiple threads blocks, which results in a substantial improvement of the running time compared to sequential CPU implementations. When designing the parallel prefix sum function, both algorithmic and hardware efficiency have been considered. The memory access patterns has also been optimized.

The overall performance has been significantly improved by managing shared memory banks conflicts. The CUDA shared memory is composed of multiple memory banks (memory modules of equal size) [2]. Accessing consecutive arrays using consecutive threads is a very

fast process because each memory bank stores 32-bit value (eg, a floating point variable). Multiple data requests from the same memory bank generate memory banks conflicts. Multiple requests might originate from the same memory address or multiple addresses can be associated to the same memory bank. The hardware device serializes memory operations when a conflict occurs and this causes all threads to wait until all memory requests are fulfilled. Serialization is avoided if all threads read from the same-shared memory address, because a broadcast mechanism is automatically triggered, an excellent high-performance method to deliver data simultaneously to many threads [1].

In the Fermi architecture the streaming multiprocessor schedules threads in groups of 32 parallel threads called warps. The parallel prefix sum function partitions data in warp-sized fragments and then independently processes these fragments using one warp per each of them. This kind of solution proves very useful when optimizing security solutions implemented by the use of the multilayered structural data like the MSDSSA presented in [6].

5. Conclusions

We believe that, in the future, harnessing the huge computational power provided by GPGPUs will have a significant impact on our everyday lives considering that more and more citizens need to access remotely resources and most of the countries have Internet access, with Finland being the first in this matter [7]. Thus, a viable solution for real-time data mining in the near future is of paramount importance for countries and citizens all over the world.
The massively parallel General Purpose Computation Graphics Processing Unit (GPGPU) and the Compute Unified Device Architecture provide viable solutions for developing parallel software that scales efficiently on the multithread processing cores. The developed software must employ a large amount of fine-grained parallelism in order to use the multithreaded structure of this architecture. Using the SIMT processing model with specific optimization techniques for data-parallel problems and solutions based on the shared memory management, is the key to obtain optimized data parallel prefix sum algorithm useful in the data mining process as a whole.

In this paper, we have analyzed solutions for optimizing the data parallel prefix sum function using the Compute Unified Device Architecture (CUDA). We have first depicted the function's design within a warp, then at the block level and finally at global level. We have benchmarked and evaluated the performance of the optimized parallel prefix sum building block in CUDA. Based on previous works in this field and on experimental results, we have developed an optimized version of the data parallel prefix sum function in CUDA, we have identified and implemented a set of solutions to improve their performance, confirmed by the obtained experimental results. The main factors that contribute to the optimization of the parallel prefix sum function in CUDA are:

- the optimization of the tasks' assignment to each thread
- the reduction of the number of used registers
- the management of the parallel task synchronization, reducing the number of the required synchronization operations in the application programming interface (API)

- the reduction of the used parallel steps, according to the amount of data processed
- the multiple thread blocks usage, which results in a substantial improvement of the running time compared to sequential CPU implementations
- we have used optimization techniques for the GPU memory management and software performance improvement of the CUDA applications [1].
- overall performance has been significantly improved by managing shared memory banks conflicts
- the data partitioning in warp-sized fragments, followed by the independent processing of these fragments using one warp per each of them.

In conclusion, the obtained experimental results confirm the efficiency of the parallel prefix sum function, that offers optimum results in different scenarios without being significantly influenced by the type of input data or the chosen binary operator, and therefore it can be implemented in a wide range of data parallel algorithms. One particular interest was to research how well the optimization techniques scale to the latest generation of general-purpose graphic processors units, like the Fermi architecture implemented in GTX480 and the previous architecture implemented in GTX280. There has been a lot of interest in the literature lately for the optimization of the parallel prefix sum function, but none of the works (to our knowledge) tried to validate if the optimizations techniques can be applied to a GPU from the Fermi architecture.

References

[1] Pirjan Alexandru, Improving software performance in the Compute Unified Device Architecture, Informatica Economica Journal, vol. 14, no. 4/December 2010, pp. 30-47.

[2] Sanders J., Kandrot E., CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, New Jersey, 2010.

[3] GPU Computing Gems Jade Edition, Wen-mei W. Hwu, Morgan Kaufmann, 2011.

[4] Harris M., Sengupta S., Owens J. D., Parallel Prefix Sum (Scan) with CUDA, in GPU Gems 3, Pearson Education, Boston, 2007.

[5] Nickolls J., Buck I., Garland M., Skadron K., Scalable parallel programming with CUDA, Queue, vol. 6, no. 2/ March-April 2008, pp. 40-53.

[6] Tăbușcă Alexandru **– „**A New Security Solution Implemented By The Use Of The Multilayered Structural Data Sectors Switching Algorithm (MSDSSA)" „Journal of Information Systems & Operations Management – Vol.4, No.2 – December 2010" ISSN 1843-4711, Universitary Publishing House, pages 164-168

[7] Tabușca, Silvia Maria - "The Internet Access as a Fundamental Right". Published in "Journal of Information Systems and Operations Management" Vol.4 No.2, Decembrie 2010, ISSN 1843-4711, Universitary Publishing House, pages 206 - 212